Palo Alto Research Center

Separating Data From Function in a Distributed File System

By Jay E. Israel, James G. Mitchell
and Howard E. Sturgis

XEROX

Separating Data From Function in a Distributed File System

by Jay E. Israel, James G. Mitchell and Howard E. Sturgis

CSL-78-5 September 1978

This paper discusses an independent file facility, one that is *not* embedded in an operating system. The *distributed file system* (DFS) is so named because it is implemented on a cooperating set of *server* computers connected by a communications network, which together create the illusion of a single, logical system for the creation, deletion, and random accessing of data. Access to the DFS can only be accomplished over the network; a computer (or, more precisely, a program running on one) that uses the DFS is called a *client*. This paper describes the division of responsibility between servers and clients. We discuss examples of situations in which a client is expected to take prescribed steps in order to achieve its intended result. The basic tool for maintaining data consistency in these situations is the *atomic property* of file actions. This is a DFS feature that protects clients from system malfunctions and from the competing activities of other clients. We have implemented an experimental system based on these concepts.

KEY WORDS AND PHRASES:

file systems, computer networks, databases, distributed systems, file structures

CR CATEGORIES:

3.73, 4.33

XEROX
PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road / Palo Alto / California 94304

1. Introduction

One of the primary functions of an operating system is providing a file storage capability (as well as allocating processor and memory resources, etc.); this paper discusses an independent file facility that is *not* embedded in an operating system. This file system operates in a distributed environment with a number of computers of several types connected by a high-bandwidth, communications network [1] and can only be accessed over this network. We have implemented an experimental version of the system.

The *distributed file system* (DFS) is so named because it is implemented on a cooperating set of *server* computers which together create the illusion of a single, logical system. The other computers in the network that use the DFS for creating, destroying, and randomly accessing files are called its *clients* (we employ the term "user" to stand only for *human* users; programs that access the DFS are always called clients).

Structuring the DFS using a set of servers is partly motivated by the economics of disk storage devices. Using rather small computers as file servers permits a wide range of file storage capacity because it is possible to expand a DFS from a single minicomputer with one disk drive to many servers, each with a number of disks. It also allows one to vary the ratio of processing power to disk capacity over a wide range to suit a variety of applications. With current disk technology, a minimal DFS (consisting of a single server with one disk drive) might hold 80 Mbytes, while a large system could consist of, say, six servers together supporting about 8000 Mbytes. This represents a range of 1:100 in storage capacity and a range of 16:1 in server computers/Mbyte of storage. Being able to vary both the amount of processing power and the raw storage capacity in a DFS means that its effective bandwidth can be matched to the usage patterns in a given environment.

The availability of high-bandwidth local communications (the Ethernet [1] in our case) and distributed computers has also motivated the design of the DFS. They make it reasonable to separate the actual processing of data from the system required to store that data reliably. This separation also contributes to reliability and system robustness in the face of failures because a hardware or software problem in one computer (whether server or client) does not affect others since the only resource they share is the communications link.

Although it can store large amounts of data, the DFS is *not* itself a data-base system: it provides no built-in access mechanisms for files other than random access to sequences of bytes and pages. In fact, it does not even support user-sensible names for files. Rather, each file is "named" by a *unique identifier* (*UID*), which is essentially a large integer. However, a directory service for maintaining a map from user-sensible names into UIDs has been implemented (as a client of the DFS rather than as an integral part of it) and is discussed in section 4.

Since the DFS is intended to support applications dealing with (possibly distributed) data bases, it provides a number of important mechanisms and guarantees which they can use. The *atomic property* for file actions is by far the most important of these guarantees: A sequence of reads and writes of some set of files (probably not all residing on the same server machine) can be performed as an *indivisible, atomic* operation. Such an action sequence is called a *transaction* and is bracketed by two special operations, *Open Transaction*, and *Close Transaction*, which are discussed more fully in section 3. All the write actions in a transaction will appear to occur simultaneously, and the information obtained by all the reads in the transaction will still be correct at the apparent time of the write. Moreover, the atomic property holds even in the presence of crashes of servers or clients; i.e., if a crash occurs, either all or none of the writes of the transaction will have been performed. A brief discussion of this mechanism is given in section 3.4; details can be found in [2].

The atomic property holds even if the files involved are accessed simultaneously by more than one transaction. Read and write locks control the accesses so that the transactions do not see inconsistent views of the data (using locks to solve this problem is discussed in detail in [3]). When one of the transactions completes, then the other, locked-out one will be able to proceed. This has the same effect as if transactions sharing files were forced to be strictly sequential in time; thus they each see consistent data. For example, assume that one transaction is attempting to move $10.00 from bank account A to B while a second is trying to withdraw funds from B. In the absence of locks, the withdrawal from B could overlap with its balance being read and then written back with an additional $10.00, an error which would have the effect of ignoring the withdrawal. But, the atomic property guarantees that no such error can occur as the result of concurrent access by multiple clients.

These controls on sharing also create the possibility of deadlock [4] between transactions accessing the same files. However, a lock set by one transaction can be broken if another transaction is trying to lock the same data and has waited too long. Thus, deadlock situations can be cleared since the atomic property allows the DFS to abort a transaction which has a broken lock. This mechanism is, however, not necessarily quick: it therefore behooves client programs for which deadlock is probable to avoid it by one of the various schemes suggested in [4]). Section 4 discusses how the lock time-out mechanism can be used by clients to maintain consistent local caches of shared data.

In the remainder of this paper we discuss the division of responsibilities between servers and clients (section 2), some of the details of transactions and accessing files using the DFS (section 3), how clients can use the basic mechanisms of the DFS to maintain local caches of shared data or to respond to aborted transactions (e.g., because of a server crash), and how a directory service is provided (section 4).

2. Clients and servers

The DFS (i.e., the set of DFS servers) provides reliable file storage, implements the atomic property, resolves concurrent access requests, deals with deadlocks (i.e., with locks that are held too long), decides which requests to honor, etc. However, it does not incorporate knowledge of application domains, or of the meaning or logical structure of files; nor is it directly usable by human users. These considerations are the responsibility of the client programs that use the DFS for their specific applications.

Splitting the client and server responsibilities in this way has several operational advantages. Here are a few examples:

Individual computers can be specialized for different tasks, yielding better system performance for a given amount of equipment.

Distributing function between DFS servers and its clients also distributes the processing cycles, which takes advantage of the processing power available in a distributed computing environment.

Decentralized control reduces the impact of malfunctions: if there is a hardware or software problem on one server, it is isolated to that machine.

The client/server split also has some longer term advantages. For example, the DFS implements files as simple sequences of bytes or pages to allow data-base application programs considerable freedom in building file structures suited to their needs. This also means that new application domains and file structure philosophies can be implemented without forcing current applications to be changed, and without having to "work around" features of built-in file structures that could hinder rather than help a new application.

Requiring clients to handle certain exceptional situations instead of the DFS means that simpler mechanisms can be built into the DFS. It is a client's responsibility to maintain its own consistency when dealing with situations such as aborted transactions. Section 4 discusses how such exceptions arise and details some techniques that clients can use to handle them. If a client fails to take care of them, the integrity of the DFS is not compromised, but the client may not achieve its intended results.

Clients are responsible not only for their specific applications, but also for maintaining the illusion of a single file system. They must communicate directly with the DFS servers holding the files that they need to manipulate. To assist in this, the DFS provides some simple file-locating facilities that can be used to find the server holding some file, given the file's UID.

Although client programs are application-specific, they will generally include a certain amount of common code for dealing with the DFS (e.g., for handling communications and some of the details of transactions). To avoid duplication of programming effort, this code

is provided by the DFS implementers in the current system. In general, however, the client software is not the responsibility of DFS. Likewise, any data kept in client machines is beyond the DFS's jurisdiction. If a client needs its own local copy (cache) of some shared information, consistency with others can be assured only if the client follows a prescribed algorithm, as described in section 4.

3. Access mechanics

This section describes the basic interactions between file servers and clients. Normally, application programs will not deal directly with the DFS, but will use some (more or less) standard underlying software to hide low-level details. For example, the underlying software will usually portray the DFS as a single file system even though it must be aware of the multiplicity of file servers. An application is not required to use the standard low-level software, however; it is free to treat the DFS at an abstract level that suits that application.

3.1 Client and server communications

Clients and servers converse by sending messages to one another (for the purposes of this paper we assume an underlying protocol for reliable, authenticated transmission of messages between machines on the network). To access a file a client sends a *request* message to a server; the outcome is communicated in the opposite direction by means of a *result* message. This one-to-one request-result interchange is the fundamental access mechanism.

Each message includes control information, which specifies the type of message and related parameters. For example, if a client wants to read file data, it would send a message specifying the action *Read Data*, and as parameters the file's UID, the index of the first byte to be read, and the number of bytes to be read. In response to *Read Data*, a server would normally send a result message specifying *Here's Data* and include the bytes read from the file.

Each result message specifies the identity of the request to which it is a response, so that the client computer can match results with requests. This permits the client to initiate new requests without waiting for previous ones to be completed. However, the actual order in which the server carries out such overlapped actions is not specified, and the client needs to be able to handle results in any order. If such parallelism is unnecessary, then the client can force sequentiality by simply waiting for the response to one request before sending the next.

Instead of returning data with a result, a server may send a *reject* response (along with a reason for not fulfilling the request). For example, the requested byte range may be

beyond the limits of the file, the requested access may be one this client is not entitled to make, etc.

Sometimes a server sends an *unsolicited* message to a client. For example, if the server breaks a lock, it will inform the client by sending an *unsolicited* message that includes the details of the situation (see section 4.2).

3.2 Details of transactions

When a client first connects to the DFS, it sends an *Open Transaction* request to some server (any one will do), which is then known as the *primary server* for that transaction. *Open Transaction* serves two purposes: as with a time-sharing system, the client presents its credentials, and the server inspects them to decide whether to permit the client access to the DFS; secondly, *Open Transaction* also begins a transaction. The server will return a result message with a unique identifier for this transaction; all further messages in the transaction will carry that identification.

Once a client has opened a transaction, it is free to request actions such as *Create File, Destroy File, Read Data*, or *Write Data.* When it is finished, the client will request a *Close Transaction*. The DFS then completes the transaction, sends a response, and disconnects from that client.

As a first step in carrying out a read action, the addressed server sets read locks on the data to be read. Similarly, as the first step in a write action, write locks are set on the data to be modified. As usual, a single datum may have multiple readers, or a single writer, but not both, and possession of a write lock allows one to read the data as well. Finally, when a server completes its part of a transaction, it clears all that transaction's read and write locks that the server controls.

3.3 Transactions with multiple servers

When a client needs to access files on more than one DFS server in a single transaction, it sends an *Add Server* request to the new server with which it needs to communicate. That server will then establish a connection with the client and will report to the primary server for the transaction (which is specified as part of the *Add Server* message) so that they can coordinate later. This enables the DFS to maintain the atomic property even when multiple servers are involved in a transaction (see section 3.4).

Once the *Add Server* request is accepted, the client can address file access requests to the new server directly (i.e., without generating any "hidden" communications among servers). Transactions involving any number of servers are set up in this way, and requests to them are essentially independent - there is no close coordination among servers during normal file accesses.

Close Transaction requests can be addressed only to the primary server. Along with *Add Server*, they are the only messages that cause communications among the servers involved in a transaction.

3.4 Completing transactions

This section describes the mechanisms by which servers maintain the atomic property of transactions in the face of crashes (a more thorough explanation is given in [2]). There are three main ideas: *stable storage, intentions lists,* and *multiple-server synchronization.*

We assume that a disk drive can be programmed to provide *stable storage*; i.e., that a page can be written to the disk so that if the system should crash while the write is in progress, either the write does not occur at all (leaving the previous contents of the page intact), or the write completes successfully. This is an atomic property for disk pages.

Although disks do not directly provide stable storage, they do provide a *weak atomic property* on which stable storage can be built using software. If some failure should occur during a disk write, the disk will be in one of three states following the failure:

(1) the write has not started yet, in which case the data on the disk is unchanged;

(2) the write completed before the failure occurred, so the new data has been correctly written on the disk;

(3) the failure occurred while the disk's write head was turned on, in which case we assume that the data on the disk is *detectably bad* (because some form of cyclic redundancy check or error-correcting and detecting code is written with every disk page and checked when the page is read).

Stable storage is bootstrapped from this weak atomic property by representing each stable "page" by two normal disk pages. Writing a page of stable storage requires writing both page copies, and the second one is not written until the first has safely completed. The pages of a pair are allocated on the disk in relatively safe places from one another (e.g., so that a single head crash won't destroy both of them), and we assume that a failure will affect at most one of a pair. If a failure should occur during the logical write, the pair of pages forming the single stable page will be in one of the following states:

(1) both pages have the same valid data;

(2) both pages have valid data, but they are not the same;

(3) one of the pages is detectably bad.

During crash recovery, each pair of pages in stable storage is read and the two are made to be the same. In case (2), we pick the first page and write its contents into the second to make a correct stable page. In case (3), the data in the good page is written into the incorrect one. Thus, the stable page has the strong atomic property desired. Reading a

stable page involves reading one of the copies; if it should be detectably bad, then the other is read instead.

It should be noted that stable storage is only needed for certain system uses such as intentions lists (see below), pointer pages to clients' files, disk allocation, etc. Thus, the overhead for stable pages involves only a very small fraction of the total storage available on a disk.

An *intentions list* is a list of actions designed so that partially carrying out the list several times and then finally completing it, is the same as performing it exactly once. Intentions lists are recorded in stable storage and can be written and erased as atomic acts. (If an intentions list is larger than one page, all trailing pages are first written, then the head page. A list does not exist unless the head page exists.)

When only a single file server is involved, a transaction is completed as a three-step process. First an intentions list which completely describes the writes of the transaction is written to stable disk storage. Next the intentions are carried out. Finally, the list is erased.

If a crash should occur, all existing intentions lists are (re)discovered during recovery from the crash. Each such intentions list is carried out and then erased. The effect of one or more crashes is that an intentions list may be partially carried out several times, then completely carried out once. In any case, the transaction completes correctly.

A simple implementation for intentions is obtained by representing files as pointer blocks, which contain a sequence of pointers to data pages. In this case, an intentions list can be a list of new pointer values. A transaction is then implemented by writing data only in newly allocated disk pages. When the client sends a *Close Transaction* request, the addresses of these new pages are collected in an intentions list.

Multiple-server synchronization is needed to avoid a situation in which one server carries out the writes of a transaction, while another aborts the transaction due to a crash. We designate the primary server as the *coordinator* and the others involved in the transaction as *workers*. The following steps complete a transaction; they are prevented from overlapping by synchronization messages at the completion of each step:

(1) The coordinator writes a list that includes the identity of each worker and marks the list *tentative*.

(2) Each worker writes an intentions list for the transaction (covering the writes occurring at that server). The worker also includes the identity of the coordinator in the list.

(3) The coordinator changes his worker list from *tentative* to *confirmed*. This effectively completes the transaction, and the coordinator can now send a response to the client that the transaction has successfully completed.

(4) The workers carry out their intentions lists and then erase them.

(5) The coordinator erases his worker list.

During crash recovery, all coordinators' lists of workers and all workers' intentions lists are located on the crashed server's disks.

When a coordinator's list is discovered, there are two cases to be considered, depending on whether the list is marked *tentative* or *confirmed*. If it is tentative, each worker of the transaction is told to delete its intentions list and then the coordinator deletes its worker list (thus aborting the transaction). If the list is marked confirmed, the above algorithm resumes at step (4) to complete the associated transaction.

When a worker's intentions list is discovered, it is simply kept until notification is received from the coordinator to either resume at step (4) or to abort.

4. Client responsibilities

4.1 Server crashes and aborted transactions

If a client machine is to provide continuous service, it must behave correctly in the presence of aborted transactions and server crashes. The effect of an abort is that all of the write requests issued by the client since beginning the transaction are lost, and all of the read and write locks are released. The atomic property guarantees that a server crash occurring before a *Close Transaction* has the same effect as an abort, while one occurring after a transaction has successfully closed has no effect.

In general, a client recovers from an aborted transaction by redoing it. This does not mean simply repeating the reads and writes of the transaction since the writes were presumably dependent on the reads, and the client may read different data the second time (all read locks have been released, so another transaction may have changed the data). Also, later read requests may have been issued on the basis of results from earlier requests. Thus, redoing a transaction includes rereading the data, repeating the intermediate calculations, and writing new results (in whatever order is appropriate for the application). Moreover, if some of the writes were based on data from human input (e.g., a teller at a bank window), then redoing it may require that the human input be resolicited.

In many cases, redoing a transaction can be made more efficient, however. Assume that the client saved the results of all the read requests and the values that were written in the aborted transaction. If rereading the data yields the same input values as previously, the client can simply write the saved results from the previous calculations and need not recompute them.

A transaction may be aborted for a number of reasons (this list is not exhaustive):

(1) The server may run out of resources and deliberately abort a transaction in an attempt to recover.

(2) The transaction may have been in existence for a long time, and a write lock may have been broken by some other transaction.

(3) Some server involved in the transaction may have crashed and recovered.

(4) The server may abort a transaction because it decides that a client has crashed when it actually has not (this might happen if communications are temporarily broken and later repaired).

If a transaction is aborted for any of these reasons, it is up to the client machine to discover the situation and repair it.

A client machine will discover that a transaction has aborted in a number of ways. In the easiest case (e.g., (1) above), the server directly informs the client machine of the abortion. Alternatively, the server may reject a requested action because it has previously aborted that transaction (cases (3) and (4) can cause this).

There is one much more difficult case: A client *C* has sent a *Close Transaction*, but before it has received a completion response, the server crashes. When the server recovers from the crash and *C* resends the *Close Transaction*, it will receive a rejection in response because the transaction is no longer around (it either aborted or was completed as part of crash recovery). *C* must then figure out which case holds, *but only by inspecting data contained in the DFS*. For example, it could have written the transaction sequence number in a private file as part of the transaction. The atomic property then guarantees that the presence of the transaction number in the private file indicates that the whole transaction completed properly; its absence means that the entire transaction aborted. To avoid this overhead, the DFS servers maintain a list of the most recently completed transactions. Thus, a client can determine if a transaction completed simply by checking to see if the transaction number is on this list. However, this list is only of finite length, so a client who waits too long (on the order of days) before asking whether some transaction has completed may receive the answer "don't know" instead of "yes" or "no".

4.2 Local caches of shared data

When data is stored remotely, it is often cheaper for a client to *cache* parts of the data locally (e.g., in its main memory or on a local disk) instead of simply making requests of the DFS for every file access. By caching we mean that the client might not only hold onto data read from the DFS, but might not necessarily write data to the DFS on every logical write (e.g., to replace two overlapping writes by a single one later). However, if a client accesses shared files, caching may introduce performance problems. For example, the atomic property is enforced by setting read and write locks as a client reads and writes

data. This mechanism has the undesirable effect that a client can hold data locked for long periods of time if he avoids closing a transaction.

A simple way to prevent a client from holding data locked indefinitely would be to abort a transaction whenever it has run longer than some predefined maximum. Unfortunately, clients may be unable to live within the confines of a transaction time limit. Instead of outlawing long-lived transactions, we have chosen to make three alterations to the basic locking mechanisms to make it efficient for clients to maintain local caches of shared data. Following descriptions of these changes, we give two examples of how clients may use the mechanisms to cache data.

The first modification to the basic mechanism is to permit the *breaking* of locks which have been set for a long time. (This is weaker than aborting a transaction.) A lock is broken when two conditions are satisfied: it has been set for a long time and another transaction attempts to set a competing lock. When a lock is broken, the transaction which set the lock is notified. This notification is recorded by the server holding the locked data, and an unsolicited message is sent to the transaction client. A client may also request a list of the current set of broken read locks.

The second modification is to permit a transaction client to clear read locks that it has set. With this change, the atomic property needs to be rephrased to read:

at the moment a transaction completes, no data has been modified by other transactions, except possibly for any data whose read lock was subsequently cleared.

When a client clears a read lock, he is, therefore, affirming that the data writes occurring during this transaction are not dependent on the data covered by the cleared read lock (Section 4.2.1 gives an example of this).

The final modification is to prevent completing a transaction that has any broken, but uncleared locks. This dual condition on broken locks allows a client to respond to an unsolicited message that a lock has been broken by clearing the lock. A client that behaves this way (see section 4.2.2 for an example) can thereby avoid repeating an entire transaction if non-critical read locks are broken. This modification is necessary to prevent a race condition between a client that checks for broken read locks before closing a transaction only to have some broken after checking, but before requesting a *Close Transaction* request.

With these mechanisms, a client machine can maintain a local copy of data for some period of time, and become informed when the original is being changed. As shown in the following examples, this mechanism improves performance by reducing the amount of information which must be transferred between server and client, but does not provide any intrinsically new function.

4.2.1 Maintaining a read-only cache

Consider a specific example of an infrequently changing data base, the locations and contents of all the freight cars on a railroad. The client machine is to maintain a display terminal which shows some portion of this information, such as the location and contents of all trains destined for Paris. It is important that the displayed information correctly represents the current contents of the data base.

If maximum transaction time limits were enforced, the client would have to repeatedly open a transaction, read all the data to be displayed, and close the transaction. With caching, the client initially reads all the desired information from the system. It then regularly checks for broken read locks by sending requests to the file servers involved and rereads any information whose locks have been broken (i.e., just the information that has changed).

4.2.2 Cache example: Updating a shared data base

Suppose that a client has a data base that changes infrequently and that the client desires to perform a transaction that first reads a substantial amount of data, then performs a long computation, and finally writes the results back to the DFS. With only a maximum time limit on transactions and no caching mechanisms, the client would proceed in three major steps:

(1) Open a transaction, read the data, and then immediately close the transaction, releasing the locks.

(2) Perform the long computation.

(3) Open a new transaction, re-read the input data and confirm that it is unchanged since read previously. If so, write the computed result back to the file system and close the transaction. If not, the client must repeat the calculation.

Caching permits the client to avoid the overhead in step (3) of re-reading the entire set of input data, to check that it has not changed. The new scenario proceeds as follows:

(1') Open a transaction and read the input data.

(2') Perform the long computation.

(3') Write the results to the file system, and then check for broken read locks. If there are none (and this is probable since we assumed infrequently changing data), then the client closes the transaction.

If there are broken read locks (it is assumed that no write locks have been broken), the client can either redo the entire transaction, or perform the following, additional steps. This increased complexity might not always be warranted, but we are assuming in this example that substantial amounts of data would have to be re-read to redo the transaction.

(4') Clear the broken locks, re-read the data covered by those locks, and perform any reads that are dependent on that data.

(5') Perform the long calculation, although there may be some shortcuts dependent on exactly which data was re-read.

(6') Issue new write requests to convert the original write requests into the new answer. Then, check again for broken read locks. If there are none, just close the transaction, otherwise repeat steps (4') through (6').

4.3 Directories

The DFS only identifies files by their UIDs, so there is no built-in way of giving a file an alphanumeric name for users. To provide such a facility, we have implemented a *directory server*. It maintains a mapping from alphanumeric identifiers to UIDs. This mapping is represented in a normal DFS file stored in the file system.

A directory server appears as an ordinary client to the DFS. So a directory server is both a server and a client at the same time. Isolating the directory service in this way means that different implementations of directories may use different file naming schemes and still share the DFS facility. Of course, a file does not need to have an alphanumeric name its UID can be used perfectly well by a client that needs to access it (this is typically the case for scratch files).

The directory server example brings out an interesting design issue. Consider a typical situation, such as creating a file, recording it in a directory, and storing data in it. From the point of view of the initiating client, these three actions should be atomic; either all of them or none of them should be done. The DFS atomic property guarantees this, but here the server sees two separate clients (the directory server and the initiating client) communicating from different computers. The issue is how to merge the two sets of actions together atomically.

The approach we have chosen is to permit several clients to participate in a single transaction. The directory server joins the initiating client's transaction by means of a special interchange of messages with the file server(s). To access the directory file, the directory server still addresses requests to the file server where the file resides, but the file service associates two clients with the same transaction. The initiating client needs to communicate with the directory server to coordinate opening and closing transactions, but this dialog is transparent to the file servers. Note also that the two clients may need quite different access privileges: the directory server has access to the directory file, but the initiating client does not.

5. Summary

We have described a stand-alone file system that is accessed over a network and consists of a number of cooperating server machines. It provides a number of mechanisms to assist remote client machines to update shared files while keeping them consistent. The most important of these mechanisms, the atomic property for transactions, allows a client to encapsulate a sequence of reads and writes as an indivisible, atomic action.

We also detailed the division of responsibilities between file servers and clients. In general, giving more responsibility to clients takes advantage of the inherent parallelism in a distributed environment. On the other hand, the DFS must keep enough authority to be able to guarantee that its mechanisms (especially the atomic property) cannot be undermined.

Finally, we described mechanisms for enabling clients to maintain local caches of shared data and gave examples of how those facilities could be used to improve performance.

6. Acknowledgements

Along with the authors, James H. Morris was a primary implementor of the DFS. Patrick Baudelaire, Ed Fiala, Butler Lampson, and Charles Simonyi participated in the design of the system. Numerous other colleagues contributed time, hardware, and software to the implementation. We would also like to thank the referees for their careful reading of the original version of this paper and for their advice in improving it.

7. References

[1] Metcalfe, R. M., and Boggs, D. R. Ethernet: distributed packet switching for local computer networks, *Comm. ACM 19* (July 1976) pp. 395-404.

[2] Lampson, B. W., and Sturgis, H. E. Crash recovery in a distributed data storage system, *Comm. ACM*, to appear.

[3] Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L. The notions of consistency and predicate locks in a database system, *Comm. ACM 11* (Nov. 1976) pp. 624-633.

[4] Holt, R. C. Some Deadlock Properties of Computer Systems, *ACM Computing Surveys 4* (Sept. 1972) pp. 179-196.

XEROX

Separating Data From Function in a Distributed File System

By Jay E. Israel, James G. Mitchell and Howard E. Sturgis

CSL-78-5

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

XEROX® is a trademark of XEROX CORPORATION Printed in U.S.A.

XEROX